A New Technique for Strictness Analysis

David A. Wright

Department of Computer Science
University of Tasmania
Hobart 7001, Australia

Internet: `wright@probitas.cs.utas.edu.au`

Abstract

Results from Unification Theory and type inference with coercions are combined to produce a new method for performing strictness analysis of functional programs. A formal deduction system is developed in which extended types are derivable for terms of the λ-calculus. These extended types contain Boolean rings describing the reduction behaviour of terms. Algorithms implementing the method are described, as well as proofs of their correctness. The method is extended to deal with recursion, polymorphism, and constants.

1 Introduction

The type of a function may be considered as an abstract description of its behaviour. Traditionally, types have described the domain and range of a function. In this paper the notion of type will be extended to include a description of the *reduction* behaviour of a function. This new notion of type will tell us whether or not a function is dependent on the *structure* of its argument, as well as providing the usual domain and range information. For the purposes of the current paper, a function will be said to be dependent on the structure of its argument if, in every reduction to head normal form of the application of the function to its argument, the *argument* is reduced to head normal form (formal definitions of these concepts are given later).

The basic idea is to introduce two function type constructors: one to represent the set of functions which are strict on their argument, and one to represent the set of functions whose result is independent of their argument. These function type constructors represent truth and falsity in a Boolean algebra. It is then natural to permit function type constructors to also include variables and Boolean expressions. Thus function type constructors may range over the full Boolean algebra. With just a little more machinery, this allows the representation of strictness information for all the conventionally typable terms (see Wright [28]).

The main application for this work is in the implementation of functional programming languages. Information about the reduction behaviour of a function is useful as this information allows the selection between alternative reduction strategies while preserving the semantics of the function. It is advantageous to be able to safely select alternative reduction strategies as some strategies have increased opportunity for the exploitation of parallelism, or are less costly to implement in a sequential fashion (see Peyton-Jones [21]).

In this paper a syntactic connection between the reduction behaviour of a function and its type is given. The approach is thus a formal one, in the sense that it is not necessary to attach any meaning (or interpretation) to functions in order to carry out deductions of types for the functions. Readers interested in semantics may wish to consult the forthcoming thesis (Wright [28]).

The structure of the paper is as follows. After some preliminary concepts are introduced in Section 2, Section 3 explains the intuition linking types and reduction, as well as the link with Unification Theory and type inference with coercions. The deduction system is presented in Section 4, then Section 5 presents an implementation of this deduction system. Section 6 considers how the deduction system may be extended to deal with recursion, polymorphism and constants. Finally, in Section 7 brief consideration is given to the time complexity of the method. Section 8 concludes the paper. The Appendix presents a proof of correctness for the main algorithm introduced in this paper.

1.1 Related work

Several other approaches for determining information about the reduction behaviour of functions exist. To place the current work in context, these alternative methods are briefly considered.

In *abstract interpretation* ([5, 7, 10]) a function is evaluated under a non-standard semantics designed so that the evaluation yields information about the function's behaviour under the standard semantics. This approach has proved a useful framework for a variety of analyses, but does have some disadvantages. Probably the most significant of these is the time complexity of the method as it is necessary to perform fixpoint iteration in order to deal with recursion. Furthermore, it is (currently) required that the *whole* fixpoint be calculated, forcing the domains being iterated over to have finite chains and adjacent elements to be finitely comparable (if an effective analysis is desired). Although I am not aware of any formal complexity analysis of higher-order strictness analysis by abstract interpretation, it appears that the method may be n-exponentially complete[1]. The framework of abstract interpretation has been successfully extended to higher-order functions (Burn *et al* [7]), non-flat domains (Wadler [24]), and polymorphism (Hughes [11]).

Another alternative is *projection analysis* (Wadler and Hughes [25]). In this approach particular functions which perform some evaluation of their arguments are used as annotations to describe how much of the structure of an argument to a function is required. The analysis may be used in both a backwards and a forwards sense. Backwards analysis involves asking the question "given that the result of a function f must be evaluated to the extent required by the function g, what function h exists which may be applied to the argument of f such that $f;g = h;f;g$?" (writing composition from left to right). Forwards analysis is simply asking this question the other way around (determing g instead of h). Projection analysis has been used to analyse non-flat domains (Wadler and

---

[1] An analysis for first-order strictness analysis appears in [10], in which it is shown that such an analysis is DEXPTIME-complete (thanks to Nick Benton and Geoffrey Burn for bringing this to my attention).

Hughes [25]) and polymorphism (Hughes [12]), but not higher-order functions. Burn [6] has investigated the relationship between projection analysis and abstract interpretation.

Barendregt, Kennaway, Klop and Sleep [3] take yet another approach. Classes of so-called *spine* redexes are identified in the head reduction of a term to head normal form. These spine redexes are particular sets of the head needed redexes of a term. Barendregt *et al* cast their work in terms of the untyped λ-calculus and define a series of algorithms to identify spine redexes in a term.

Kuo and Mishra [15] have developed a method for performing strictness analysis using the techniques of type inference. As in the work presented here, their method has the advantage that fixpoint iteration is avoided. However, their type language fails to give information obtainable in some semantic-based approaches about some useful examples involving recursion (see [15] and Section 6.3 of this paper). Unlike the present work, the type system of Kuo and Mishra is *non-structural* in the sense that coercions (or constraints) between structured types (such as functional types) cannot be broken up into coercions between the components of these structured types. This complicates their type inference algorithm.

In the approach described in this paper, results from Unification Theory [23] are combined with the theory of type inference with coercions, to produce a decidable analysis of strictness information for functional languages. The result is an expressive and natural type language for expressing strictness information, which has the additional advantage of having a structural nature.

2 Preliminaries

Definition 1 (λ-terms)
Let $X = \{v_0, v_1, \ldots\}$ be a set of *term variables*, then the set Λ of *λ-terms* is inductively defined to be the smallest set containing X which is closed under function application and abstraction, namely

- if $M \in \Lambda$ and $N \in \Lambda$ then $MN \in \Lambda$ and
- if $x \in X$ and $N \in \Lambda$ then $\lambda x.N \in \Lambda$.

Following the usual conventions, application associates to the left, and the scope of an abstraction is as far to the right as possible. Parentheses will often be omitted where these conventions make clear the intended meaning. Also, a term of the form $(\lambda x_1.\lambda x_2.\cdots\lambda x_n.N)$ will often be written as $(\lambda x_1 x_2 \cdots x_n.N)$.

Definition 2
In a λ-term $(\lambda x.N)$ the object λx is the *binder* of the term, and x in λx is the *binding occurrence* of x. A variable x occurs *bound* in a term M if M has a subterm $(\lambda x.N)$ and x occurs in N, in which case the term N is the *scope* of this binding occurrence of x. A variable x occurs *free* in M if it is a subterm of M, and occurs outside the scope of any binding occurrence of x. The set of all free variables of a term M is denoted by FV(M). The set of all bound variables of a term M is denoted by BV(M).

Definition 3
Let $M[x := N]$ denote the result of replacing each free occurrence of x in M by N. The three rules of *λ-reduction* are:

(α-reduction) if $y \notin \mathrm{FV}(N) \cup \mathrm{BV}(N)$, then $(\lambda x.N) \xrightarrow{\alpha} (\lambda y.N[x:=y])$,

(β-reduction) $(\lambda x.N)M \xrightarrow{\beta} N[x:=M]$ and

(η-reduction) if $x \notin \mathrm{FV}(N)$, then $(\lambda x.Nx) \xrightarrow{\eta} N$.

If a λ-term M has the form of any of the left-hand sides of these rules, then M is a (α,β or η)-*redex*. If a term contains no β-redexes then it is in β-*normal form* (and similarly for η- and $\beta\eta$-normal forms). Let the reflexive, transitive closure of $\xrightarrow{\beta}$ be denoted by $\overset{\beta}{\twoheadrightarrow}$ (and similarly for $\overset{\alpha}{\twoheadrightarrow}$ and $\overset{\eta}{\twoheadrightarrow}$), and the reflexive, transitive and symmetric closure of $\xrightarrow{\beta}$ be denoted by $\overset{\beta}{=}$ (similarly $\overset{\alpha}{=}$ and $\overset{\eta}{=}$), this relation being called β-*conversion*.

From now on λ-terms will be considered modulo α-conversion (i.e. $\Lambda/\overset{\alpha}{=}$). Also, only β-reduction will be considered in the following and so "reduction" will mean β-reduction, "redex" will mean β-redex and "normal form" will mean β-normal form.

Definition 4
A subterm N of a term M is at the *head* of M iff

- $M \equiv N$, or
- $M \equiv \lambda x.N'$ and N is at the head of N', or
- $M \equiv N_1 N_2$ and N is at the head of N_1.

Suppose M is not in normal form. The *leftmost* redex of M is the redex whose binder is to the left of the binder of every other redex in M. The leftmost redex, R, of M is a *head* redex of M iff R is at the head of M. M is in *head normal form* if it has no head redex. The *head reduction path* of a term M is a sequence of reduction steps in which every redex which is reduced is a head-redex.

Suppose $M \overset{\beta}{\twoheadrightarrow} N$, then the *descendents* of some subterm M' of M can be found in N (if any exist), by marking M' and following it through the reduction from M to N. See Klop [14], pp18–19 for a formal definition of descendent in terms of labelled reduction. Any descendent of a redex, R, is itself a redex, and is called a *residual*.

Definition 5
Suppose $R \equiv (\lambda x.N_1)N_2$, then $\lambda x.N_1$ *head needs its argument*, N_2, if a descendent of N_2 occurs at the head of some term on every reduction path of R to head normal form.

Definition 6 (Barendregt et al [3])
Suppose R is a redex of M, then R is *head-needed* in M if every reduction sequence of M to head normal form reduces a residual of R.

Terms which cannot be reduced to head normal form are identified with the symbol $\perp$. A function f is *strict* on its argument if $f\perp = \perp$. The process of determining whether a function is strict is referred to as *strictness analysis*. Similarly the process of determining whether a redex is head needed is referred to as *head neededness analysis*. By Proposition 5.1 of Barendregt et al [3] (due to H. Mulder), strictness analysis is equivalent to head neededness analysis, since a function is strict on its argument iff the function head needs its argument.

Now some notation concerning substitutions of various kinds. Let Vars(x) for some object x denote the set of all variables of that object (in the case of a λ-term, M, $\mathrm{Vars}(M) = \mathrm{FV}(M) \cup \mathrm{BV}(M)$). Composition of substitutions will be written from left to right, $(f;g)(x) = g(f(x))$. Write $R[x:=y]$ for the substitution which is equal to R at every point except x, at which point it is equal to y. A substitution for a set of variables will often be extended to operate over a term by applying the substitution to each variable in the term. Finally, write $\mathrm{Dom}(R)$ for $\{x|x \neq R(x)\}$.

3 Types and Reduction

Consider the identity function, $(\lambda x.x)$. Traditionally this function is given the type $\alpha \rightarrow \alpha$ to reflect the fact that it *is* a function and that its argument type, α, is the same as its result type, α. This function certainly head needs its argument (Definition 5) since $(\lambda x.x)M \xrightarrow{\beta} M$, for every $M \in \Lambda$. To incorporate into the type the fact that it head needs its argument, the type of the identity function will be written as

$$\alpha \Rightarrow \alpha.$$

This now tells us that the term $(\lambda x.x)$ is a function, that its argument type is identical to its result type and that it head needs its argument. (Read this type as "α head needed-to α" or "α strict-to α"). Note that this type contains both strictness information as well as all the information of a conventional type, and so is a true extension of conventional types.

Similarly, consider the constant function $(\lambda x.y)$. This function is traditionally given the type $\alpha \rightarrow \beta$ (with α and β possibly distinct) to indicate that it is a function and that its argument and result types are independent of each other. The value of this function, when given an argument, is surely independent of the value or structure of this argument as $(\lambda x.y)M \xrightarrow{\beta} y$, for every $M \in \Lambda$. This information is added to the type of $(\lambda x.y)$ by writing the type as

$$\alpha \twoheadrightarrow \beta.$$

This type states that $(\lambda x.y)$ is a function, that its result and argument types are independent of each other and that for its value to be determined, when given an argument, it is not necessary that the value of its argument be determined. (Read this type as "α constant-to β").

Although it would be convenient if the above notation were sufficient, it is necessary to consider more complex functions and the types which should be assigned to them.

The function $(\lambda f.fx)$ head needs its argument as $(\lambda f.fx)M \xrightarrow{\beta} Mx$. Traditionally $(\lambda f.fx)$ is given the type $(\alpha \rightarrow \beta) \rightarrow \beta$. The difference here is that the argument of $(\lambda f.fx)$ has an explicit functional type. In the reduction sequence $(\lambda f.fx)(\lambda y.y) \xrightarrow{\beta} (\lambda y.y)x \xrightarrow{\beta} x$, it can be seen that the functional argument to $(\lambda f.fx)$ head needs its argument and therefore the type of $(\lambda f.fx)$ in this instance may be reasonably written as $(\alpha \Rightarrow \alpha) \Rightarrow \alpha$. In contrast, in the reduction sequence $(\lambda f.fx)(\lambda x.y) \xrightarrow{\beta} (\lambda x.y)x \xrightarrow{\beta} y$, the functional argument of $(\lambda f.fx)$ has a constant type and so the type of $(\lambda f.fx)$ in this instance may be written as $(\alpha \nrightarrow \beta) \Rightarrow \beta$.

There are two things to note about these two types for $(\lambda f.fx)$. Firstly, both of these types for $(\lambda f.fx)$ tell us that it head needs its argument *no matter what form that argument takes*. Hence, the second function type constructor in both types is the head needed function type constructor. Secondly, both types disagree on the first function type constructor. While it is easy to decide which is correct given the argument to $(\lambda f.fx)$, the problem arises as to what type should be given to $(\lambda f.fx)$ when no argument is present. It is of course still desirable that if an argument is eventually given to $(\lambda f.fx)$ then the appropriate one of the two types for $(\lambda f.fx)$ given above should be derivable. This problem is solved by the introduction of *variable* function type constructors, which are denoted by an arrow with a subscripted number. Thus the type of $(\lambda f.fx)$ can be written, for some variable function type constructor $\rightarrow_1$, as

$$(\alpha \rightarrow_1 \beta) \Rightarrow \beta.$$

This type states that

- $(\lambda f.fx)$ is a function,
- that its argument is also a function,
- that this functional argument takes an argument which is possibly independent of its result type,
- that the result type of the functional argument is the same as the result type of $(\lambda f.fx)$,
- that the head neededness property of its functional argument is unknown, and
- that $(\lambda f.fx)$ head needs its argument.

($(\alpha \rightarrow_1 \beta) \Rightarrow \beta$ may be read "a function which takes a functional argument of type α variable-one-to β which is head needed-to β").

As another example, consider the function $(\lambda x.f(gx))$. Ordinarily this function would be assigned the type $\alpha \rightarrow \gamma$, assuming $(g : \alpha \rightarrow \beta)$ and $(f : \beta \rightarrow \gamma)$. To determine whether $(\lambda x.f(gx))$ head needs its argument, the manner in which the variable x is used in the function must be found. It is easy to see, upon examination of $(\lambda x.f(gx))$, that this function will head need its argument if both f and g head need their arguments.

To capture this kind of information, the function type constructors are extended to a *Boolean algebra* of function type constructors. So function type constructors may now be Boolean expressions built from the function type constructors introduced above. Thus the type $\alpha\ (\rightarrow_1 \wedge \rightarrow_2)\ \gamma$ should be assigned to $(\lambda x.f(gx))$, assuming $(g : \alpha \rightarrow_1 \beta)$ and $(f : \beta \rightarrow_2 \gamma)$. (The type $\alpha\ (\rightarrow_1 \wedge \rightarrow_2)\ \gamma$ may be read "α variable-one *and* variable-two to β").

It is instructive to see how the choice of functions for f and g in the above example affect its head neededness. For example, let f and g both be the identity function, then $\rightarrow_1$ and $\rightarrow_2$ should both be instantiated to $\Rightarrow$ and so the expression $(\rightarrow_1 \wedge \rightarrow_2) = (\Rightarrow \wedge \Rightarrow)$ should "evaluate" to $\Rightarrow$, giving the expected type $\alpha \Rightarrow \alpha$ for $(\lambda x.f(gx))$. (This is of course the behaviour to be expected of $\wedge$). Alternatively, if g is the constant function $(\lambda x.y)$, then $\rightarrow_1$ should be instantiated to $\not\rightarrow$. In this case $(\lambda x.f(gx))$ is constant on its argument (g ignores x in (gx)) and so $(\rightarrow_1 \wedge \rightarrow_2) = (\not\rightarrow \wedge \rightarrow_2)$ should evaluate to $\not\rightarrow$.

Similarly, functions such as $(\lambda x.fx(gx))$ have more complex types, this function having the type $\alpha\ (\rightarrow_1 \vee (\rightarrow_2 \wedge \rightarrow_3))\ \gamma$, assuming $(g\colon \alpha \rightarrow_3 \beta)$ and $(f\colon \alpha \rightarrow_1 \beta \rightarrow_2 \gamma)$. This type conveys the following facts about $(\lambda x.fx(gx))$:

- it is a function,
- its argument and result types are possibly unrelated,
- it head needs its argument if the function f head needs its first argument, and
- it head needs its argument if f head needs its second argument and the function g head needs its sole argument.

From these examples it can be seen that a *Boolean algebra* of function type constructors is required to describe the dependency relations between subterms. In this Boolean algebra, $\Rightarrow$ plays the role of the distinguished element 1 of a traditional Boolean algebra and $\not\rightarrow$ plays the role of the 0 element. Although the negation operator, $\neg$, has not yet entered into any of the examples, an important use of this will be made later.

It is worthwhile taking a moment to consider the utility of this notation. Every λ-abstraction of a typed term will introduce a new function of type τ_1 b τ_2, where τ_2 is the type of the term, τ_1 is the type of the variable being abstracted over and b is a Boolean algebra expression over function type constructors as described above, which denotes the useage of the abstracted variable in the term. Thus to determine whether a particular argument to a function is head needed, all that must be done is to see if the Boolean algebra expression b is (or is equivalent to) $\Rightarrow$. Furthermore, if τ_1 and τ_2 are both functional types, it is possible for the head neededness of the argument(s) to the first argument to affect the strictness of subsequent arguments to the term. In this case there will be variable function type constructors common to both τ_1 and τ_2.

3.1 Unification Theory, Arrow Expressions and Types

Since the types used in this paper have Boolean expressions of arrows in them it will be necessary to perform unification of these Boolean expressions while doing type inference. Fortunately, results from Unification Theory are available which make this task decidable (Martin and Nipkow [18]).

A Boolean algebra is a set B containing distinguished elements 0 (the zero element) and 1 (the unit element) under the operations $\wedge$, $\vee$ and $\neg$, where for all x, y and $z \in B$ the following hold:

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
x \wedge x & = & x & x \vee y & = & y \vee x & \neg 1 & = & 0 \\
x \wedge 1 & = & x & (x \vee y) \vee z & = & x \vee (y \vee z) & \neg 0 & = & 1 \\
(x \vee y) \wedge z & = & (x \wedge z) \vee (y \wedge z) & x \vee 0 & = & x & \neg(x \vee y) & = & \neg x \wedge \neg y \\
x \wedge y & = & y \wedge x & x \vee x & = & x & \neg(x \wedge y) & = & \neg x \vee \neg y \\
(x \wedge y) \wedge z & = & x \wedge (y \wedge z) & x \vee 1 & = & 1 & & & \\
x \wedge 0 & = & 0 & & & & & &
\end{array}$$

Let $\overset{\mathrm{BA}}{=}$ denote the "=" operation over Boolean algebras above.

Function type constructors are built from a set of basic function type constructors called *arrows*:

Definition 7 (Arrows)
The set of *arrows* (Δ) is $\Delta = \Delta_g \cup \Delta_v$, where

- The *ground arrows* (Δ_g) are $\{\Rightarrow, \nrightarrow\}$
- The *variable arrows* (Δ_v), also called *arrow variables*, are $\{\rightarrow_1, \rightarrow_2, \ldots\}$

The arrow $\Rightarrow$ is called the *head needed* (or *strict*) arrow and the arrow $\nrightarrow$ is called the *constant* arrow. The set of function type constructors will also be called *arrow expressions* and the letters $\mathrm{b}, \mathrm{b}_1, \mathrm{b}_2 \ldots$ will be used to range over them. Following Martin and Nipkow [18], write $\mathcal{T}(B, \wedge, \vee, \neg)$ for the Boolean algebra over B generated by $\wedge$, $\vee$ and $\neg$.

Definition 8 (Arrow Expressions)
The Boolean algebra of arrow expressions is $\mathcal{B}_\Delta = \mathcal{T}(\Delta, \wedge, \vee, \neg)$. The set of *variable arrow expressions* is $\mathcal{B}_{\Delta_v} = \mathcal{B}_\Delta - \Delta_g$.

In the following, arrow expressions and variable arrow expressions will be considered modulo $\overset{\mathrm{BA}}{=}$.

It will prove useful in the following sections to extend this notion of equality between arrow expressions.

Definition 9
Let b_1 and b_2 be arrow expressions, then an *arrow constraint* is a term of the form $\mathrm{b}_1 \leq \mathrm{b}_2$. An arrow constraint, $\mathrm{b}_1 \leq \mathrm{b}_2$, is *valid* iff there exists a substitution $R\colon \Delta_v \rightarrow \mathcal{B}_\Delta$ such that $\mathrm{b}_2 \overset{\mathrm{BA}}{=} R(\mathrm{b}_1)$. An arrow expression, b_1, is *substitution equivalent* to an arrow expression, b_2, (written $\mathrm{b}_1 \overset{\mathrm{SUB}}{=} \mathrm{b}_2$) iff $\mathrm{b}_1 \leq \mathrm{b}_2$ and $\mathrm{b}_2 \leq \mathrm{b}_1$ are valid arrow constraints.

If the arrow expressions are considered modulo substitution equivalence, then they "collapse" to a three point domain, as shown by the following lemma. This three point domain fits well with the intuition that a function of type $\sigma \Rightarrow \tau$ head needs its argument σ, a function of type $\sigma \nrightarrow \tau$ is constant on its argument and that a function of type $\sigma\ \mathrm{b}\ \tau$ ($\mathrm{b} \in \mathcal{B}_{\Delta_v}$) may head need its argument. (It should be emphasized that in the last case the value b of $\sigma\ \mathrm{b}\ \tau$ will still give useful information about the function.) A function of type $\sigma\ \mathrm{b}\ \tau$ ($\mathrm{b} \in \mathcal{B}_{\Delta_v}$) should be implemented using a "lazy" reduction strategy, see Peyton-Jones [21].

Write $[\mathrm{b}]^E$ for the equivalence class of arrow expressions for b generated by relation E.

Lemma 1
$(\mathcal{B}_\Delta / \overset{\mathrm{SUB}}{=}) = \{[\Rightarrow]^{\mathrm{SUB}}, [\nrightarrow]^{\mathrm{SUB}}, [\rightarrow_i]^{\mathrm{SUB}}\}$.

Proof
For all $\mathrm{b} \in \Delta_g$, $\mathrm{b} \overset{\mathrm{SUB}}{=} \mathrm{b}'$ iff $\mathrm{b}' \overset{\mathrm{BA}}{=} \mathrm{b}$. For all $\mathrm{b} \notin \Delta_g$, it is easy to show by induction on b

```
$BUnify(\mathrm{b}_1, \mathrm{b}_2) \overset{\mathrm{def}}{=} BUnify'((\mathrm{b}_1 \wedge \neg\mathrm{b}_2) \vee (\neg\mathrm{b}_1 \wedge \mathrm{b}_2))$
$BUnify'(f(x_1, \ldots, x_n))$
        $\overset{\mathrm{def}}{=}$ if $n = 0$
           then if $f(x_1, \ldots, x_n) \overset{\mathrm{BA}}{=} \not\rightarrow$ then Id
                else fail
           else let $G = BUnify'(f(\not\rightarrow, x_2, \ldots, x_n) \wedge f(\Rightarrow, x_2, \ldots, x_n))$
                in $G[x_1 := ((\neg f(\Rightarrow, G(x_2), \ldots, G(x_n))) \wedge x_1) \vee f(\not\rightarrow, G(x_2), \ldots, G(x_n))]$
```

Figure 1: The Algorithm for unifying Boolean Arrow Expressions

that b $\overset{\mathrm{SUB}}{=} \rightarrow_i$. Finally, although it is easy to find an R such that b $\overset{\mathrm{BA}}{=} R(\rightarrow_i)$, for b $\in \Delta_g$, there is no S such that $\rightarrow_i \overset{\mathrm{BA}}{=} S(\mathrm{b})$. □

From Martin and Nipkow [18] comes the useful result that unification of Boolean expressions is unitary and decidable, that is there exists an effective algorithm for determining the most general unifier of two Boolean expressions (if it exists), and furthermore this most general unifier is unique. The algorithm for Boolean unification based on variable elimination is reproduced from [18] in Figure 1. As a notational convenience, note that a Boolean expression of n variables ($n \geq 0$), b, may be written as a function, $f(x_1, \ldots, x_n)$, of its variables.

Lemma 2
Either algorithm *BUnify* succeeds with the most general unifier of its arguments, or it reports failure.

Proof
See Martin and Nipkow [18]. □

Definition 10
Let $\tau_v = \{t_0, t_1, \ldots\}$ be a set of *type variables.*The set of *Boolean strictness types*, T, is the smallest set containing τ_v closed under function type construction using the Boolean arrow function type constructors, $\mathcal{B}_\Delta$.

The standard unification algorithm for types ([22]) can then be extended by an orthogonal combination with the algorithm for arrow unification, to produce an algorithm to unify Boolean strictness types.

3.2 Orderings for Arrows and Types

Consider a function such as $(\lambda x.\lambda y.f(gx)(gy))$, which would conventionally be assigned the type $\alpha \rightarrow \alpha \rightarrow \beta$ (assuming $(g : \alpha \rightarrow \gamma)$ and $(f : \gamma \rightarrow \gamma \rightarrow \beta)$). To give $(\lambda x.\lambda y.f(gx)(gy))$ a Boolean strictness type, suppose $(g\colon \alpha \rightarrow_1 \gamma)$ and $(f\colon \gamma \rightarrow_2 \gamma \rightarrow_3 \beta)$, then a first attempt might be $\alpha\ (\rightarrow_1 \wedge \rightarrow_2)\ \alpha\ (\rightarrow_1 \wedge \rightarrow_3)\ \beta$. However, while this is a legal Boolean strictness type for $(\lambda x.\lambda y.f(gx)(gy))$, it is not the most general type that can be found. Currently, this type states that

- $(\lambda x.\lambda y.f(gx)(gy))$ is a function of two arguments,
- the type of its two arguments must be identical (including their head neededness properties!),
- the result and argument types are possibly unrelated.

A more general type assignable to $(\lambda x.\lambda y.f(gx)(gy))$ is $\alpha' \ (\rightarrow_1 \wedge \rightarrow'_3) \ \alpha'' \ (\rightarrow_2 \wedge \rightarrow''_3) \ \beta$, where any type substituted for α' must be a renaming instance of the type substituted for α (and similarily for α''). This notion of "renaming instance" will be formally defined below. (For clarity, renamed variables will often be written dashed, this makes it easier to identify the "parent" variables). Any head neededness properties substituted for $\rightarrow'_3$ and $\rightarrow''_3$ must also be instances of $\rightarrow_3$. To summarise, this example has the following *constraints*:

- $\alpha \rightarrow_3 \gamma \leq \alpha' \rightarrow'_3 \gamma'$, and
- $\alpha \rightarrow_3 \gamma \leq \alpha'' \rightarrow''_3 \gamma''$,

which can also be expressed as

- $\alpha \leq \alpha'$, $\rightarrow_3 \leq \rightarrow'_3$, $\gamma \leq \gamma'$, and
- $\alpha \leq \alpha''$, $\rightarrow_3 \leq \rightarrow''_3$, $\gamma \leq \gamma''$.

One thing to note about this is that $\leq$ is not anti-monotonic over the first argument of a functional type (cf. Mitchell [20]). This is because it is desirable to maintain the maximum head neededness information in the types assigned to occurrences of functions.

Definition 11
Write $\sigma \overset{\mathrm{BA}}{=} \tau$ iff σ and τ are identical except that corresponding arrow expressions are related by $\overset{\mathrm{BA}}{=}$.

- Let σ and τ be types, then a *type constraint* is a term of the form $\sigma \leq \tau$.
- A type constraint, $\sigma \leq \tau$, is *valid* iff there exists a renaming $R : \tau_v \rightarrow \tau_v$ and a substitution on arrows $S : \Delta_v \rightarrow \mathcal{B}_\Delta$, such that $\tau \overset{\mathrm{BA}}{=} R; S(\sigma)$.

A term of the form $x \leq y$, for any arrow expressions x and y or any types x and y, will be called a *constraint* and is a *valid constraint* if it is a valid arrow or type constraint. A *constraint set* is simply a set of constraints as defined above and a constraint set C is *valid* iff all its elements are valid. Constraints $\sigma \leq \sigma'$, $\mathrm{b} \leq \mathrm{b}'$ and $\tau \leq \tau'$ are *compatible* iff there exist substitutions $R : \tau_v \rightarrow \tau_v$ and $S : \Delta_v \rightarrow \mathcal{B}_\Delta$ such that $\sigma' \overset{\mathrm{BA}}{=} R; S(\sigma)$, $\mathrm{b}' \overset{\mathrm{BA}}{=} S(\mathrm{b})$ and $\tau' \overset{\mathrm{BA}}{=} R; S(\tau)$. A constraint set may entail other constraints as captured by the following definition:

Definition 12
For any constraint set C, the *entailment* relation, $\rhd$, is defined to be the smallest relation satisfying:

1. $C \cup \{x \leq y\} \rhd x \leq y$,

2. $C \rhd x \leq x$,

3. $C \cup \{\sigma\ \mathrm{b}\ \tau \leq \sigma'\ \mathrm{b}'\ \tau'\} \rhd \begin{cases} \sigma \leq \sigma' \\ \mathrm{b} \leq \mathrm{b}' \\ \tau \leq \tau' \end{cases}$,

4. If $\sigma \leq \sigma'$, $\mathrm{b} \leq \mathrm{b}'$ and $\tau \leq \tau'$ are compatible, then $C \cup \{\sigma \leq \sigma', \mathrm{b} \leq \mathrm{b}', \tau \leq \tau'\} \rhd \sigma\ \mathrm{b}\ \tau \leq \sigma'\ \mathrm{b}'\ \tau'$, and

5. $C \cup \{x_1 \leq x_2, x_2 \leq x_3\} \rhd x_1 \leq x_3$.

Lemma 3
If C is valid, then $C \rhd x \leq y$ implies $x \leq y$ is valid.

Proof
Straightforward by induction over $\rhd$. □

In the following, write $C \rhd C'$ if C entails everything that C' does, that is $C' \rhd x \leq y$ implies $C \rhd x \leq y$. Two constraint sets C and C' are *equivalent* (written $C \simeq C'$) iff $C \rhd C' \rhd C$. Finally, a substitution S *respects* a valid constraint set C iff $S(C)$ is valid.

4 A Deduction System for Strictness Analysis

As in traditional type systems, if a λ-term has free variables, then its type will be determined by what types are assumed for those free variables. It will thus be necessary to associate with each term a set of assumptions about the types of its free variables, in order to understand the type assigned to the term. (Denote this *assumption set* by $A: X \times T$).

In addition to the assumption set of types for free variables of a term, if an abstraction of a term variable is to be performed, then it will be necessary to know how the term from which the variable is being abstracted *uses* (or head needs) that variable. Thus, for each free variable of a term a mapping giving an arrow expression describing its use in the term is required. This last is called a *variable useage function*, and will be denoted by $V: X \rightarrow \mathcal{B}_\Delta$ in the following. The variable useage function which maps all term variables to $\not\twoheadrightarrow$ will be denoted by $V_{\not\twoheadrightarrow}$.

A *typing statement* is a quintuple of a constraint set C, an assumption set A, a variable useage function V, a λ-term M and a type σ, written as

$$C, A \vdash V, M : \sigma.$$

The variable useage function is written to the right of the turnstile as it is a *result* of the deduction system, as will be seen shortly.

The deduction system for the terms introduced so far is given in Figure 2. A brief discussion of the unfamiliar aspects of the system is warranted.

In rule VAR, the variable useage function is $V_{\not\twoheadrightarrow}$ except for the point at x at which the value is $\Rightarrow$, since if an abstraction of the term x with respect to the variable x is performed, then the identity function is obtained and the identity function head needs its argument. In addition, any type with more specific head neededness information than the assumption for x in A may be assigned to x (as long as C entails this). This corresponds

$$\text{VAR} \qquad C, A_x \cup \{x : \sigma\} \vdash V_{\nrightarrow}[x := \Rightarrow], x : \tau \qquad (C \rhd \sigma \leq \tau)$$

$$\text{APP} \quad \frac{C, A \vdash V_1, N_1 : \sigma \mathrm{~b~} \tau \quad C, A \vdash V_2, N_2 : \sigma}{C, A \vdash V, N_1 N_2 : \tau} \quad \left(V = \bigcup_{x \in X} \{x := V_1(x) \vee (\mathrm{b} \wedge V_2(x))\}\right)$$

$$\text{ABS} \quad \frac{C, A_x \cup \{x : \sigma\} \vdash V[x := \mathrm{b}], N : \tau}{C, A \vdash V[x := \nrightarrow], \lambda x.N : \sigma \mathrm{~b~} \tau}$$

Figure 2: The Deduction System for Strictness Analysis

to the intuition that a term x may be influenced by the *context* in which it appears in the term. Examples of this were presented earlier in Section 3.2.

Rule APP is conventional apart from the treatment of the variable useage functions. The construction of V from V_1 and V_2 may be informally justified as follows. Any free variable which is head needed by the functional term in the application will be head needed in the application. Free variables in the argument term will be head needed by the application if the argument term is head needed by the functional term and if the argument term itself head needs them.

In the case of the ABS rule, since an abstraction closes the scope of the term variable being abstracted, any further abstraction of that variable will result in a function which does not have the variable free within it, thus the variable useage function should have its entry for the abstraction variable set to $\nrightarrow$ in the resultant typing statement.

Thus at each stage of the deduction of a typing statement the variable useage function is completely specified, which justifies its placement to the right of the turnstile.

Definition 13
For any valid constraint set C, a typing statement $C, A \vdash V, M : \sigma$ is a *well-typing* iff it is the conclusion of a deduction using the rules of Figure 2.

At this stage only a rather limited class of terms has been considered. In Section 6 this class is extended to a more practical language.

Example 1
A well-typing for $\lambda x.x$ is $\{\alpha \leq \beta\}, \emptyset \vdash V_{\nrightarrow}, \lambda x.x : \alpha \Rightarrow \beta$. This yields the expected result that the identity function head needs its argument. Note the constraint $\alpha \leq \beta$, which allows the argument to $\lambda x.x$ to have its head neededness information specialised to another context.

Example 2
Consider the function *Twice* $\equiv \lambda fx.f(fx)$. A well-typing for *Twice* is

$$\{\alpha_1 \leq \alpha_1', \alpha_3 \leq \alpha_1', \alpha_1 \leq \alpha_2', \alpha_2 \leq \alpha_2', \alpha_2 \leq \alpha_2'', \rightarrow_1 \leq \rightarrow_1', \rightarrow_1 \leq \rightarrow_1''\},$$
$$\emptyset \vdash V_{\nrightarrow}, \mathit{Twice} : (\alpha_1 \rightarrow_1 \alpha_2) \Rightarrow \alpha_3(\rightarrow_1' \wedge \rightarrow_1'')\ \alpha_2''.$$

This type for *Twice* indicates that f is head needed (as expected) and that x is head needed in *Twice* if it is head needed by *both* occurrences of f. This latter is important if the argument substituted for f is a function like $\lambda ag.ga$, which only head needs a if g does.

Write $V_1 \overset{\mathrm{BA}}{=} V_2$ iff $\forall x \in X.V_1(x) \overset{\mathrm{BA}}{=} V_2(x)$ and write $A_1 \overset{\mathrm{BA}}{=} A_2$ iff $\forall i,j \in \{1,2\}$, $x\!:\sigma \in A_i$ implies $x\!:\tau \in A_j$ and $\sigma \overset{\mathrm{BA}}{=} \tau$.

Definition 14
A typing statement $C', A' \vdash V', M\!:\tau$ is an *instance* of a typing statement $C, A \vdash V, M\!:\sigma$ iff there exists a substitution S which respects C such that

- $A'|_{\mathrm{FV}(M)} \overset{\mathrm{BA}}{=} S(A|_{\mathrm{FV}(M)})$,
- $V' \overset{\mathrm{BA}}{=} S(V)$,
- $\tau \overset{\mathrm{BA}}{=} S(\sigma)$, and
- $C' \simeq S(C)$.

Lemma 4 (Instances)
If $C, A \vdash V, M : \sigma$ is a well-typing, then every instance, $C', A' \vdash V', M : \tau$, of $C, A \vdash V, M\!:\sigma$ is a well-typing.

Proof
By induction on the structure of M.

$M \equiv x$: Since $C, A_x \cup \{x\!:\sigma'\} \vdash V_{\rightarrow}[x := \Rightarrow], x : \sigma$ is a well-typing (that is $C \rhd \sigma' \leq \sigma$ and C is valid), and $C' \simeq S(C)$ (by Definition 14), the result follows by rule VAR.

$M \equiv N_1 N_2$ **or** $M \equiv \lambda x.N$: Both cases follow by induction and use of rule APP or ABS, respectively.

□

Note that the instance relation on typing statements is a *pre-order.*

5 Algorithms for Strictness Analysis

In defining an algorithm to implement the deduction system it is desirable that a most general representative well-typing of a term is computed. This will enable all other well-typings of the term to be obtained from the most general representative (see Lemma 4). In this section such an algorithm is described and its correctness is proved. The algorithms in this section are presented in a style similar to the work of Mitchell [20] and Fuh and Mishra [9] on ordinary type inference with coercions.

A deduction of a well-typing for a term may be seen as a tree with leaf nodes being instances of rule VAR, unary nodes instances of rule ABS and binary nodes instances of rule APP. The algorithm works by inserting most general constraints at the leaves (this is achieved by using type and arrow variables which have not been used before), and then

Freshen$(\alpha) \overset{\text{def}}{=} \beta$, where β is a new type variable,
Freshen$(\sigma \text{ b } \tau) \overset{\text{def}}{=}$*Freshen*$(\sigma)$ b$'$ *Freshen*(τ)
where b$'$ = b if b $\in \Delta_g$, else b$' = \rightarrow_i$, for $\rightarrow_i$ a new arrow variable.

Figure 3: The Algorithm for computing Renaming Instances

equating types at binary nodes (unification) while ensuring that all leaf inequalities are maintained (expansion).

To create most general constraints two operations are useful. *Allnew*, simply replaces all arrow expressions by new arrow variables and all type variables by new type variables. For example, *Allnew*$(\alpha\ (\rightarrow_1 \vee \rightarrow_2)\ \beta) = \gamma \rightarrow_3 \epsilon$. *Freshen*, is defined in Figure 3.

Lemma 5

For all types σ, $\sigma \leq$ *Freshen*(σ) and *Freshen*$(\sigma) \leq \sigma$ are valid.

Proof

By induction on the structure of σ. For the basis case, the result follows immediately since *Freshen* simply renames type variables. For the induction case, note that *Freshen* preserves ground arrow types, so the result follows by Lemma 1 and by the induction hypotheses. □

Note that *Freshen*$(\sigma) \overset{\text{BA}}{\neq} \sigma$ as *Freshen* renames type variables.

The operation of expansion ensures that the conditions of Definition 11 are satisfied, that is that a constraint set C is valid. It does this by returning a valid version of the constraint set and the substitution required to make the original constraint set valid. Let $[\alpha]_C = \{\beta | C \rhd \alpha \leq \beta \vee C \rhd \beta \leq \alpha\}$ and $[\sigma \text{ b } \tau]_C = \{[\alpha]_C | \alpha \in \text{Vars}(\sigma \text{ b } \tau)\}$. The algorithm for maintaining the validity of constraint sets is given in Figure 4.

Lemma 6

If C is a valid constraint set, R is any substitution and $(C', R') =$ *Expand*$(R(C))$ succeeds, then

1. $R; R'$ respects C,
2. C' is valid and
3. $R; R'(C) \simeq C'$.

Proof

All three of these may be shown by induction and by observing that *Expand* converts each invalid constraint into a valid constraint (using Lemmas 5 and 2), and that R' is the (in order) collection of all substitutions required for the conversion. (The first case of *Expand* simply decomposes functional types, which is a validity preserving transformation.) □

Algorithm *Type* (see Figure 5) implements the strictness analyser. The Appendix to this paper contains two theorems which together establish the correctness of algorithm *Type* with respect to the deduction system.

```
$Expand(C \cup \{\sigma_1\ \mathrm{b}_1\ \tau_1 \le \sigma_2\ \mathrm{b}_2\ \tau_2\}) \stackrel{\mathrm{def}}{=} Expand(C \cup \{\sigma_1 \le \sigma_2, \mathrm{b}_1 \le \mathrm{b}_2, \tau_1 \le \tau_2\})$
$Expand(C \cup \{\alpha \le \sigma\ \mathrm{b}\ \tau\}) \stackrel{\mathrm{def}}{=}$ if $[\alpha]_C \in [\sigma\ \mathrm{b}\ \tau]_C$ then fail
                                   else let $R = \mathrm{Id}[\alpha := Allnew(\sigma\ \mathrm{b}\ \tau)]$ and
                                            $(C', R') = Expand(R(C \cup \{\alpha \le \sigma\ \mathrm{b}\ \tau\}))$
                                       in
                                            $(C', R; R')$
$Expand(C \cup \{\sigma\ \mathrm{b}\ \tau \le \alpha\}) \stackrel{\mathrm{def}}{=}$ if $[\alpha]_C \in [\sigma\ \mathrm{b}\ \tau]_C$ then fail
                                   else let $R = \mathrm{Id}[\alpha := Freshen(\sigma\ \mathrm{b}\ \tau)]$ and
                                            $(C', R') = Expand(R(C \cup \{\sigma\ \mathrm{b}\ \tau \le \alpha\}))$
                                       in
                                            $(C', R; R')$
$Expand(C \cup \{\mathrm{b}_1 \le \mathrm{b}_2\})$, where $\mathrm{b}_1 \in \Delta_g$
                                   $\stackrel{\mathrm{def}}{=}$ let $R = BUnify(\mathrm{b}_1, \mathrm{b}_2)$ and
                                          $(C', R') = Expand(R(C))$
                                   in
                                          $(C', R; R')$
$Expand(C) \stackrel{\mathrm{def}}{=} (C, \mathrm{Id})$, if $C$ is valid
```

Figure 4: The Algorithm for Expansion of Constraint Sets

```
$Type(C, A, x) \stackrel{\mathrm{def}}{=} (C \cup \{A(x) \le \sigma\},\ V_{\not\rightarrow}[x := \Rightarrow],\ \sigma,\ \mathrm{Id})$, where $\sigma = Freshen(A(x))$
$Type(C, A, \lambda x.N)$
    $\stackrel{\mathrm{def}}{=}$ let $(C', V, \tau, S) = Type(C, A_x \cup \{x : \alpha\}, N)$, where $\alpha \notin \mathrm{Vars}(A) \cup \mathrm{Vars}(C)$
       in
          $(C',\ V[x := \not\rightarrow],\ S(\alpha)\ V(x)\ \tau,\ S)$
$Type(C, A, N_1 N_2)$
    $\stackrel{\mathrm{def}}{=}$ let
          $(C_1, V_1, \sigma_1, S_1) = Type(C, A, N_1)$ and
          $(C_2, V_2, \sigma_2, S_2) = Type(C_1, S_1(A), N_2)$ and
          $R_1 = Unify(S_2(\sigma_1), \sigma_2 \rightarrow_i \alpha)$, where $\alpha$ and $\rightarrow_i$ are new variables, and
          $(C', R_2) = Expand(R_1(C_2))$ and
          $V' = R_1; R_2(\bigcup_{x \in X} \{x := S_2(V_1(x)) \vee (\rightarrow_i \wedge V_2(x))\})$
       in
          $(C',\ V',\ R_1; R_2(\alpha),\ S_1; S_2; R_1; R_2)$
```

Figure 5: The Algorithm for Typing Terms

6 Polymorphism, Constants and Recursion

In this section the class of terms is extended to encompass a more practical programming language. In particular, recursion is introduced through a *fixpoint* operator, polymorphism is introduced through the Milner style polymorphic LET construct and constants are introduced into the type and term languages.

The complete deduction system is collected in Figure 6. Due to limitation of space, the additions required to algorithm *Type* in order to implement these extensions will not be given here (see Wright [28]).

6.1 Polymorphism

Parametric polymorphism is introduced into the deduction system using Milner's LET construct (Milner [19]). To simplify the presentation, the method of Leiss [16] is followed instead of the more traditional quantified types of Milner. The LET construct is added to the set of terms with the following clause:

- $x \in X$, $N_1, N_2 \in \Lambda$ implies let $x = N_1$ in $N_2 \in \Lambda$.

For details of the semantics of this term see Milner [19]. The extension to allow strictness analysis of this term then proceeds as follows.

Firstly, the relation of *generic instance* (Milner [19], Leiss [16]) is required. This must operate on both type and arrow variables, instead of just on type variables.

Definition 15
Write $C|\{x_1,\ldots,x_n\}$ for $\{x \leq y | C \rhd x \leq y \wedge \{x,y\} \cap \{x_1,\ldots,x_n\} \neq \emptyset\}$, then write $\sigma^C \preceq_A \tau^{C'}$ ($\tau^{C'}$ is a *generic instance* of σ^C) iff there exists a substitution R which respects C such that

- $R(\sigma) \overset{\mathrm{BA}}{=} \tau$,
- $R(C)|\mathrm{Vars}(\tau) \simeq C'$, and
- $\mathrm{Dom}(R) = \mathrm{Vars}(\sigma) - \mathrm{Vars}(A)$.

The rule for the polymorphic LET construct then follows as in Leiss [16]. The major variation is the treatment of the variable useage function (see rule LET of Figure 6). Clearly the polymorphic variable being defined cannot appear in N_1, this is enforced by requiring the deduction of a type for N_1 to be made under the assumption A_x. As a result, the value of the variable useage function at x should be $\not\rightarrow$. The variable useage function built as a result of the deduction is similar to the case of the APP rule (as expected).

Secondly, since the assumption set may now contain polymorphic term variables, a new rule is required, rule PVAR (see Figure 6). It is illuminating to compare this rule with rule VAR and rule CON.

6.2 Constants

Constants may be introduced into types and terms in a straightforward manner. Firstly, let $K = \{\ldots, \mathbf{bool}, \mathbf{int}, \ldots\}$ be the set of *constant types*, then extend types (Definition 10) with the clause:

- $\kappa \in K$ implies $\kappa \in T$.

Secondly, extend the inductive definition of terms:

- $c \in \kappa$, $\kappa \in T$ implies $c \in \Lambda$.

Some common constants and their types appear in Figure 7. Constants exhibiting parametric polymorphism are dealt with in a similar way to LET-bound variables. Note that since constants can never be bound in an abstraction or LET construct, the variable useage function is $V_{\nrightarrow}$. Rule CON of Figure 6 incorporates this approach to constants into the deduction system.

Since the present analysis is concerned with determining how much of the structure of its argument a function requires, making the assumption $1 : \mathbf{int}$ does not result in any loss of strictness information.

6.3 Recursion

In strictness analysers such as [7, 10], the treatment of recursion involves iterating over finite chains to find a fixpoint of the abstract function. The present approach avoids this costly technique.

Recursion is modelled in the λ-calculus by using terms such as Turing's fixpoint combinator, Θ (see Barendregt [1]), which has the property that for any term $M \in \Lambda$, $\Theta M \overset{\beta}{\twoheadrightarrow} M(\Theta M)$. Unfortunately, in the type system upon which most functional languages are currently based no type can be found for terms which contain self-application of variables, as is the case for Θ (and other fixpoint combinators). One way to avoid this problem is to switch to a complete type discipline (such as *intersection types*, see Barendregt *et al* [2]), but in this paper the more usual approach to types has been used, as type inference is decidable for these systems. As a notational convenience, write fix $x.N$ for $\Theta(\lambda x.N)$.

Consider, a term of the form: fix $F.\lambda a.Fa$, which is clearly strict on its argument as (fix $F.\lambda a.Fa)M = \bot$, for *all* $M \in \Lambda$. In contrast, consider the term: fix $F.\lambda ab.Ka(Fab)$, where $K \equiv \lambda xy.x$. The value of this term is clearly only strict on its first argument (whose binding variable is a), but is independent of its second argument. This is easily verified since $\varnothing, \varnothing \vdash V_{\nrightarrow}, \lambda xy.x : \alpha \Rightarrow \beta \nrightarrow \alpha$ is a well-typing for K.

There are thus two things which influence the strictness of a fix variable (x in fix $x.N$) on its arguments. The first is the strictness properties of the body of the fix expression (that is, N in fix $x.N$), since if this expression is strict (constant) on an argument then the fix expression will be strict (constant) on that argument. The second is that the fix variable of a fix expression represents the fix expression itself (thus implementing the recursion). If the fix variable is head needed by the fix expression, then non-termination will result (and the deduction system can report that it has found a syntactically determinable non-terminating computation). Of course, in this case the fix expression is strict

$$\text{VAR} \qquad C, A_x \cup \{x\colon \sigma\} \vdash V_{\nrightarrow}[x := \Rightarrow], x\colon \tau \qquad (C \rhd \sigma \leq \tau)$$

$$\text{APP} \qquad \frac{C, A \vdash V_1, N_1\colon \sigma \text{ b } \tau \quad C, A \vdash V_2, N_2\colon \sigma}{C, A \vdash V, N_1 N_2\colon \tau} \qquad \left(V = \bigcup_{x\in X} \{x := V_1(x) \vee (\text{b} \wedge V_2(x))\}\right)$$

$$\text{ABS} \qquad \frac{C, A_x \cup \{x\colon \sigma\} \vdash V[x := \text{b}], N\colon \tau}{C, A \vdash V[x := \nrightarrow], \lambda x.N\colon \sigma \text{ b } \tau}$$

$$\text{FIX} \qquad \frac{C, A_x \cup \{x\colon \sigma \uparrow \text{b}\} \vdash V[x := \text{b}], N\colon \tau}{C, A \vdash V[x := \nrightarrow], \text{fix } x.N\colon \tau} \qquad (C \rhd \tau \leq \sigma)$$

$$\text{PVAR} \qquad C, A \cup \{x\colon \sigma^{C'}\} \vdash V_{\nrightarrow}[x := \Rightarrow], x\colon \tau \qquad \left(C \rhd C' \rhd \sigma \leq \tau \right)$$

$$\text{LET} \qquad \frac{\begin{array}{l} C, A_x \cup \{x\colon \sigma_1^{C_1}, \ldots, x\colon \sigma_n^{C_n}\} \vdash V_1[x := \text{b}], N_1\colon \tau \\ C, A_x \vdash V_2[x := \nrightarrow], N_2\colon \sigma \end{array}}{C, A \vdash V, \text{let } x = N_2 \text{ in } N_1\colon \tau} \qquad \left(\begin{array}{l} V = \bigcup_{x\in X} \{x := V_1(x) \vee (\text{b} \wedge V_2(x))\} \\ \sigma^C \preceq_{A_x} \sigma_i^{C_i}, \quad 1 \leq i \leq n \end{array}\right)$$

$$\text{CON} \qquad C, A_c \cup \{c\colon \sigma^{C'}\} \vdash V_{\nrightarrow}, c\colon \tau \qquad \left(\begin{array}{c} \sigma^{C'} \preceq_{A_c} \rho^{C''} \\ C \rhd C'' \rhd \rho \leq \tau \end{array}\right)$$

Figure 6: The Deduction System with Fixpoints, Polymorphism and Constants

Constant	Type
if_then_else_	**bool** $\Rightarrow \alpha_1 \ \rightarrow_1 \ \alpha_2 \ (\neg \rightarrow_1) \ \alpha_3$, where $\{\alpha_3 \leq \alpha_1, \alpha_3 \leq \alpha_2\}$
$(\neg)$	**bool** $\Rightarrow$ **bool**
$(=), (\leq), \ldots$	**int** $\Rightarrow$ **int** $\Rightarrow$ **bool**
$\ldots, -1, 0, 1, \ldots$	**int**
$(+), (-), \ldots$	**int** $\Rightarrow$ **int** $\Rightarrow$ **int**

Figure 7: Some useful constants and their types

on its arguments. On the other hand, the fix expression is independent of occurrences of arguments to any instance of the fix variable which the fix expression is itself independent of. In general, where the strictness of an argument is not determined by the fix expression, it will be determined by the head neededness of the fix variable.

This intuition motivates the following definition.

Definition 16
The *strictified* by b form of a type σ, written $\sigma \uparrow \mathrm{b}$, is defined as follows:

- $\alpha \uparrow \mathrm{b} = \alpha$ and
- $(\tau\ \mathrm{b}'\ \tau') \uparrow \mathrm{b} = \begin{cases} \tau\ \mathrm{b}'\ (\tau' \uparrow \mathrm{b}), \text{ if } \mathrm{b}' \in \Delta_g \\ \tau\ \mathrm{b}\ (\tau' \uparrow \mathrm{b}), \text{ otherwise} \end{cases}$.

A further complication is that a fix variable may be used in a strictness context which is an instance of that of the strictness context of the fix expression. The relevent rule is rule FIX of Figure 6.

In the following examples it is useful to refer to Figure 7 to derive the types associated with particular constants.

Example 3
Suppose $H \equiv \text{fix } F.\lambda abc.$ if c then $Fab(\neg c)$ else a, then a well-typing for H is: $\varnothing, \varnothing \vdash V_{\rightsquigarrow}, H : \alpha \Rightarrow \beta \rightarrow_1 \mathbf{bool} \Rightarrow \alpha$. This type tells us that H is strict on a and c and lazy on b. Note that in this expression the fix variable is head needed if the then part of the conditional is head needed and so this is the head neededness of b. Also, the head neededness of a is $\rightarrow_1 \vee (\neg \rightarrow_1) \overset{\mathrm{BA}}{=} \Rightarrow$. This is the most information that we can expect from any static analysis.

Example 4
A well-typing for *napply* $\equiv$ fix $F.\lambda nfx.$if $n = 0$ then x else $F\ (n-1)\ f\ (fx)$, is:

$$\varnothing, \varnothing \vdash V_{\rightsquigarrow}, \mathit{napply} \colon \mathbf{int} \Rightarrow (\alpha \rightarrow_2 \alpha)\ (\neg \rightarrow_1)\alpha\ (\rightarrow_1 \vee \rightarrow_2)\ \alpha.$$

This type tells us that *napply* is strict on n, strict on x if the then branch of the conditional is ever taken or if f is strict on its argument and that it is strict on f if the else branch of the conditional is ever taken.

Example 5
Consider the function $H \equiv \text{fix } F.\lambda xyz.$if $z = 0$ then $x + y$ else $Fyx(z-1)$, which also appears in Kuo and Mishra [15]. As explained by Kuo and Mishra, their technique is unable to detect all the strictness information in this example. Using the approach of this paper, the well-typing deducible for this example is the optimal one: $\varnothing, \varnothing \vdash V_{\rightsquigarrow}, H\colon \mathbf{int} \Rightarrow \mathbf{int} \Rightarrow \mathbf{int} \Rightarrow \mathbf{int}$.

7 Complexity

Recent results have shown that the problem of determining whether or not an ML expression is typable is DEXPTIME-complete (see Mairson [17] and Kfoury, Tiuryn and

Urzyczyn [13]). Furthermore, Wand and O'Keefe [26] have shown that the general problem of finding a typing in the presence of coercions between base types is NP-complete. However, in the case that the coercions form a tree they show that the problem is solvable in "low-order" polynomial time. This is fortunate for the current approach since the constraints between arrow expressions are tree structured (when considered as a partial order).

Another result of relevance is that of Martin and Nipkow [18] which shows that Boolean ring unification is exponential in the number of variables in the expressions to be unified. Since the number of constraints introduced in typing a function is proportional to the number of occurrences of term variables, and it is usual for this to be a small number, solving the constraints and performing the Boolean ring unifications should be efficient in most situations.

Although further investigation is required, it is conjectured that in practice these results will not preclude the use of the current technique. (Type inference techniques have seen practical use for many years now as their exponential behaviour is in the length of individual function definitions and not in the number of function definitions, see Mairson [17] or Kfoury, Tiuryn and Urzyczyn [13]). Certainly the technique offers hope of greater efficiency than current abstract interpretations involving higher-order functions, since it is known that *first-order* strictness analysis by abstract interpretation is DEXPTIME-complete (also see the footnote on page 37 of Bloss [4], in which it is noted that an implementation of a restricted higher-order strictness analysis proved to have an impractical time complexity).

8 Conclusion

In this paper the deduction of strictness information for functional programs has been examined and a formal system developed which determines such information from untyped terms. Connections between Unification Theory, type inference with coercions and the reduction behaviour of terms have been revealed.

Although this paper has concentrated on the analysis of strictness information, there is no reason why the framework of type theory may not be used to describe and explore other properties of functional programs. A further open area is the general relationship between the frameworks of abstract interpretation and type theory.

Wright [28] shows how the technique introduced in this paper may be split into two phases, the first being type inference resulting in explicitly typed terms in the style of Mitchell [20], and the second being the strictness analysis phase which annotates these explicitly typed terms with strictness information.

Algorithms implementing the core of the deduction system were presented. These algorithms employ an algorithm for performing unification of Boolean rings of arrow expressions (based on the work of Martin and Nipkow [18]). Significant extensions were given to the deduction system which allow the treatment of important features of a practical functional language. An extension of the method to allow strictness analysis of arbitrary user-defined algebraic data types has also been developed, but space does not allow for its presentation.

Strictness analysis via type inference has thus been shown to be possible (Kuo and

Mishra [15], Wright [27]) and moreover has proved to be an especially natural way of performing the analysis. The completely uniform treatment of higher-order functions is particularly pleasing, as well as the generality of the method.

Acknowledgements

I would like to thank Clem Baker-Finch and Tony Dekker for their time and many useful discussions. I would also like to thank Nick Benton, Geoffrey Burn, Phil Collier, Andrew Partridge and the anonymous referees for their comments on drafts of this paper.

Appendix

The following theorems show the correctness of algorithm Type. Their structure is similar to the theorems appearing in the Appendix to Fuh and Mishra [9].

Theorem 1 (Syntactic Soundness)
If C is a valid constraint set and $(C', V, \sigma, S) = \mathit{Type}(C, A, M)$ succeeds, then $C', S(A) \vdash V, M : \sigma$ is a well-typing.

Proof
By induction on the structure of M.

$M \equiv x$ Since $S = \mathrm{Id}$, $A(x) \leq \sigma$ is valid (using Lemma 5), $C' \simeq C \cup \{A(x) \leq \sigma\} \triangleright A(x) \leq \sigma$ and C' is valid (since C and $A(x) \leq \sigma$ are valid), the result follows immediately by use of rule VAR.

$M \equiv \lambda x.N$ By the induction hypothesis, $(C', V[x := \mathrm{b}], \tau, S) = \mathit{Type}(C, A_x \cup \{x : \alpha\}, N)$ succeeds and $C', S(A_x \cup \{x : \alpha\}) \vdash V[x := \mathrm{b}], N : \tau$ is a well-typing. Further, since $(C', V[x := \nrightarrow], S(\alpha) \mathrm{~b~} \tau, S) = \mathit{Type}(C, A, \lambda x.N)$, the result follows directly from rule ABS.

$M \equiv N_1 N_2$ By the induction hypothesis $C_1, S_1(A) \vdash V_1, N_1 : \sigma_1$ and $C_2, S_1; S_2(A) \vdash V_2, N_2 : \sigma_2$ are well-typings. Since $C_2 \simeq S_2(C_1) \simeq S_1; S_2(C)$ are valid (by the definition of well-typing, Definition 13), S_2 and $S_1; S_2$ respect C_1 and C (respectively).

Now, $R_1; R_2(C_2) \simeq S_2; R_1; R_2(C_1) \simeq S_1; S_2; R_1; R_2(C) \simeq C'$ are all valid constraint sets since $R_1; R_2$ respects C_2 (using Lemma 6), so the instance lemma (Lemma 4) applies:

- $C', S_1; S_2; R_1; R_2(A) \vdash S_2; R_1; R_2(V_1), N_1 : S_2; R_1; R_2(\sigma_1)$ is a well-typing and
- $C', S_1; S_2; R_1; R_2(A) \vdash R_1; R_2(V_1), N_2 : R_1; R_2(\sigma_2)$ is a well-typing, such that $R_1; R_2(\sigma_2 \rightarrow_i \alpha) \overset{\mathrm{BA}}{=} R_1; R_2(\sigma_1)$.

The result then follows by using these two well-typings as antecedants to an appropriate instance of the APP rule.

□

Theorem 2 (Syntactic Completeness)
If $C'', A'' \vdash V'', M : \sigma''$ is a well typing and is an instance of well-typing $C, A \vdash V, M : \sigma$ with instance substitution R, then $(C', V', \sigma', S) = \mathit{Type}(C, A, M)$ succeeds and $C'', A'' \vdash V'', M : \sigma''$ is an instance of $C', S(A) \vdash V', M : \sigma'$.

Proof
By induction on M.

$M \equiv x$ Clearly $(C \cup \{A(x) \leq \sigma'\}, V_{\nrightarrow}[x := \Rightarrow], \sigma', \mathrm{Id}) = \mathit{Type}(C, A, x)$ succeeds. Let S be a substitution such that $S(\sigma') \overset{\mathrm{BA}}{=} \sigma''$ and $\forall x \notin \mathrm{Vars}(\sigma').S(x) \overset{\mathrm{BA}}{=} R(x)$, then $S(C \cup \{A(x) \leq \sigma'\}) \simeq C''$ (since $R(C) \simeq C''$), $S(A(x)) \overset{\mathrm{BA}}{=} A''(x)$ and S respects $C \cup \{A(x) \leq \sigma'\}$. Therefore $C'', A'' \vdash V_{\nrightarrow}[x := \Rightarrow], x : \sigma''$ is an instance of $C \cup \{\sigma \leq A(x)\}, A \vdash V_{\nrightarrow}[x := \Rightarrow], x : \sigma'$, with instance substitution S.

$M \equiv \lambda x.N$ By the induction hypothesis, $(C', V', \tau_1, S) = \mathit{Type}(C, A_x \cup \{x : \alpha\}, N)$ succeeds and $C'', A''_x \cup \{x : \tau_2\} \vdash V, N : \tau_3$ is an instance of $C', S(A_x \cup \{x : \alpha\}) \vdash V', N : \tau_1$, where $\sigma'' \overset{\mathrm{BA}}{=} \tau_2 \; V''(x) \; \tau_3$ and $\sigma' \overset{\mathrm{BA}}{=} S(\alpha) \; V'(x) \; \tau_1$. The result then follows immediately by rule ABS and the definition of *Type*.

$M \equiv N_1 N_2$ Since we have a deduction of $C'', A'' \vdash V'', N_1 N_2 : \sigma''$, by rule APP we must also have deductions of $C'', A'' \vdash V''_1, N_1 : \tau'' \; \mathrm{b}'' \; \sigma''$ and $C'', A'' \vdash V''_2, N_2 : \tau''$, where $V'' = \bigcup_{x \in X}\{x := V''_1(x) \vee (\mathrm{b}'' \wedge V''_2(x))\}$. By the induction hypotheses, both $(C_1, V_1, \sigma_1, S_1) = \mathit{Type}(C, A, N_1)$ and $(C_2, V_2, \sigma_2, S_2) = \mathit{Type}(C_1, S_1(A), N_2)$ succeeds. Also by the induction hypothesis, there exists T_1 and T_2 such that

1. $A''|_{\mathrm{FV}(N_1N_2)} \overset{\mathrm{BA}}{=} S_1; T_1(A|_{\mathrm{FV}(N_1N_2)}) \overset{\mathrm{BA}}{=} S_1; S_2; T_2(A|_{\mathrm{FV}(N_1N_2)})$,
2. $C'' \simeq T_1(C_1) \simeq T_2(C_2) \simeq S_2; T_2(C_1)$,
3. $\tau'' \; \mathrm{b}'' \; \sigma'' \overset{\mathrm{BA}}{=} T_1(\sigma_1)$ and $\sigma'' \overset{\mathrm{BA}}{=} T_2(\sigma_2)$, and
4. $V''_1 \overset{\mathrm{BA}}{=} T_1(V_1)$ and $V''_2 \overset{\mathrm{BA}}{=} T_2(V_2)$.

Using (1), $\tau'' \; \mathrm{b}'' \; \sigma'' \overset{\mathrm{BA}}{=} S_2; T_2(\sigma_1) \overset{\mathrm{BA}}{=} T_1(\sigma_1)$ and since R is the most general unifier of $S_2(\sigma_1)$ and $\sigma_2 \rightarrow_i \alpha$, there exists a T_3 such that $T_2 = R; T_3$. Now, $\tau'' \overset{\mathrm{BA}}{=} R; T_3(\sigma_2)$ and $\sigma'' \overset{\mathrm{BA}}{=} R; T_3(\alpha)$, so using Lemma 6 there exists T_4 such that $T_3 = R'; T_4$ and thus

- $A''|_{\mathrm{FV}(N_1N_2)} \overset{\mathrm{BA}}{=} S_1; S_2; R; R'; T_4(A|_{\mathrm{FV}(N_1N_2)})$,
- $R; R'; T_4(C_2) \simeq S_2; R; R'; T_4(C_1) \simeq C''$,
- $\sigma'' \overset{\mathrm{BA}}{=} R; R'; T_4(\alpha)$, and
- $V'' \overset{\mathrm{BA}}{=} R; R'; T_4(V')$.

□

References

[1] Barendregt, H.P. *The Lambda-Calculus: its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, second edition, 1984.

[2] Barendregt H.P., M. Coppo and M. Dezani-Ciancaglini. A Filter Lambda-Model and the Completeness of Type Assignment. *Journal of Symbolic Logic*, 48(4):931–940, December 1983.

[3] Barendregt H.P., J.R. Kennaway, J.W. Klop and M.R. Sleep. Needed Reduction and Spine Strategies for the Lambda-Calculus. Technical report, Centre for Mathematics and Computer Science, May 1986.

[4] Bloss, A. Update Analysis and the Efficient Implementation of Functional Aggregates. In *Functional Programming and Computer Architecture*, pages 26–38. ACM, 1989.

[5] Burn, G.L. *Abstract Interpretation and the Parallel Evaluation of Functional Languages.* PhD thesis, Imperial College, March 1987.

[6] Burn, G.L. A Relationship between Abstract Interpretation and Projection Analysis. In *Principles of Programming Languages*, pages 151–156. ACM, 1990.

[7] Burn G.L., C.L. Hankin and S. Abramsky. Strictness Analysis for Higher-Order Functions. *Science of Computer Programming*, 7:249–278, 1986.

[8] Fuh Y-C and P. Mishra. Type Inference with Subtypes. In *European Symposium on Programming*, pages 94–114 Springer Verlag, LNCS 300, 1988.

[9] Fuh Y-C and P. Mishra. Polymorphic Subtype Inference: Closing the Theory-Practice Gap. In *Theory and Practice of Software Development*, pages 167–183. Springer Verlag, LNCS 352, 1989.

[10] Hudak P. and R.Young. Higher-Order Strictness Analysis in Untyped Lambda-Calculus. In *Principles of Programming Languages*, pages 97–109. ACM, 1986.

[11] Hughes, R.J.M. Abstract Interpretation of Polymorphic Functions. In *Functional Programming*, pages 31–40. Springer-Verlag, 1989.

[12] Hughes, R.J.M. Projections for Polymorphic Strictness Analysis. In *Category Theory and Computer Science*, pages 82–100. Springer Verlag, LNCS 389, 1989.

[13] Kfoury A.J., J. Tiuryn and P. Urzyczyn. ML Typability is DEXPTIME-Complete. In *CAAP'90*, pages 206–220. Springer Verlag, LNCS 431, 1990.

[14] Klop, J.W. *Combinatory Reduction Systems.* PhD thesis, State University of Utrecht, 1980.

[15] Kuo T-M and P. Mishra. Strictness Analysis: A New Perspective based on Type Inference. In *Functional Programming and Computer Architecture*, pages 260–272. ACM, 1989.

[16] Leiss, H. Polymorphic Recursion and Semi-Unification In *CSL'89*, pages 211–224. Springer Verlag, LNCS 440, 1989.

[17] Mairson, H.G. Deciding ML Typability is Complete for Deterministic Exponential Time. In *Principles of Programming Languages*, pages 382–401. ACM, 1990.

[18] Martin U. and T. Nipkow. Boolean Ring Unification—the story so far. *Journal of Symbolic Computation*, 7:275–293, 1989.

[19] Milner, R. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[20] Mitchell, J.C. Coercion and Type Inference (Summary). In *Principles of Programming Languages*, pages 175–185. ACM, 1984.

[21] Peyton-Jones, S.L. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[22] Robinson, J.A. A Machine-Orientated Logic based on the Resolution Principle. *Journal of the ACM*, 12:23–41, 1965.

[23] Siekmann, J.H. Unification Theory. *Journal of Symbolic Computation*, 7:207–274, 1989.

[24] Wadler, P.L. Strictness Analysis on Non-flat Domains (by Abstract Interpretation over Finite Domains). In S. Abramsky and C.L. Hankin, editors, *Abstract interpretation of declarative languages.* Ellis Horwood Limited, 1987.

[25] Wadler P.L. and R.J.M. Hughes. Projections for Strictness Analysis. In *Functional Programming Languages and Computer Architecture*, pages 385–407. Springer-Verlag, LNCS 274, 1987.

[26] Wand M. and P. O'Keefe. On the Complexity of Type Inference with Coercion. In *Functional Programming and Computer Architecture*, pages 293–297. ACM, 1989.

[27] Wright, D.A. Strictness Analysis Via (Type) Inference. Technical Report 89/3, University of Tasmania, September 1989.

[28] Wright, D.A. *Type Inference and the Reduction Behaviour of Functional Programs.* PhD thesis, University of Tasmania. (In preparation).